一种低开销软硬件混合的细粒度内存行为分析方法

陈荔城 崔泽汉 包云岗 陈明宇 黄永兵 谭光明

摘 要：内存行为分析是进行内存系统调度、体系结构及应用访存性能等优化的基础，而细粒度的内存行为分析能够标识内存系统性能瓶颈的源头，并为优化提供丰富的语义信息。常用的内存行为分析手段包括插桩、模拟器、硬件计数器等，但它们分别存在开销大，准确性不足，无法提供详细信息等问题。本文提出了一种软硬件混合的细粒度内存行为分析方法，能够对程序的完整访存序列进行函数级和对象级分析。硬件方面使用 HMTT 卡监控系统访存请求，软件方面采用二进制插桩方式来获取函数入口、出口信息，通过导出内核页表及对象内存分配信息来得到每个对象的内存空间信息。实验结果表明，本文提出的方法能够以较低的开销，准确地获取真实系统上的函数及对象级的访存序列。

关键词：内存行为分析 HMTT 软硬件混合 函数 对象

1 引言

随着单个处理器芯片上集成的核个数不断增加，内存作为多核系统的共享资源，将受到越来越大的访问压力，内存系统已成为多核系统性能的主要瓶颈[1]。因此准确地获取程序在真实系统上运行的访存踪迹（trace，即访存地址序列），并基于此对程序执行过程中的访存行为进行分析变得至关重要，这是进行内存系统调度及结构优化的基础。内存行为分析的手段有很多种，包括：编译驱动、动态插桩、模拟器、及硬件性能计数器（Hardware Performance Counter）等。编译驱动通过在程序编译或链接的过程中插桩植入用于收集访存序列的代码，这样在运行时就不需要其它处理，所以运行开销较低。但是它需要对程序进行重新编译或链接，这可能需要较长的时间。而且这种方法无法对那些没提供源码的程序进行处理。动态插桩采用即时编译（Just In Time, JIT）技术在程序运行时动态插入桩代码， 能够得到包含丰富信息的访存序列，但是动态插桩的运行开销比较大，不适合用于那些运行时间较长的程序。文献[2]指出：使用动态插桩技术（如 Pin[1]）来获取对象级的访存行为，即使只进行 10%的数据采样，带来的运行开销高达程序原始运行时间的 30~80 倍。模拟器是另外一种可用于收集丰富信息的内存分析手段，但是模拟器得到的访存序列不是真实机器上产生的，因此存在准确性问题，而精确模拟的开销非常大。硬件性能计数器通过统计内存相关事件来获取程序在真实机器上的访存行为，但是硬件事件计数器只能得到一些统计数据，而无法得到详细的访存序列。

表1. 不同内存分析方法的对比

	准确	详细	低开销	页表访存
插桩	√	√	×	×
模拟器	*	√	×	×
硬件性能计数器	√	×	√	*
编译支持	√	√	√	×
混合软硬件方法	√	√	√	√

表 1 总结了这些内存分析方法的对比。这里需要指出的是，上述方法只能收集那些由于

[1] 美国英特尔公司和科罗拉多大学开发的一款程序分析工具软件

访问的数据在高速缓存（Cache）中缺失（Miss）导致的访存请求，而不能收集真实系统上那些由于旁路转换缓冲缺失（Transaction Lookaside Buffer Miss, TLB Miss） 导致的页表访存（page memory walks）。因为旁路转换缓冲缺失导致的系统性能下降可以达到 5~14%[3]，所以在真实多核系统上进行页表访存行为分析变得越来越重要。以前研究旁路转换缓冲行为基本是通过将收集的访存序列输入到旁路转换缓冲模拟器中来实现，并基于模拟结果来指导优化。但是旁路转换缓冲模拟器存在准确性问题（与真实系统的旁路转换缓冲实现存在差异)。而本文提出的软硬件混合的内存分析方法，能够监控整个系统的所有访存请求，包括操作系统访存[4]、直接内存存取（Direct Memory Access, DMA）访存[5] 、页表访存等。通过监控内核中的页表状态更新，我们能够准确识别出页表访存，并将其对应到程序的对象。

为了进一步分析访存序列对应的上层语义信息，本文提出了对访存地址序列进行更细粒度的函数级和对象级的划分。

为了得到函数（调用）的入口、出口信息，我们通过直接修改内存中的进程映像（process image），在目标函数的入口及出口位置分别插入一条额外的特殊标签（tag）访存指令。这些标签访存与程序正常的访存都将被混合访存踪迹收集工具（Hybrid Memory Trace Toolkit，HMTT）[4]硬件侦听到。通过解析不同标签访存与函数入口、出口之间的映射关系，就能够将底层的访存踪迹与高层的函数执行序列关联起来，从而实现对访存踪迹的函数级别分割。

为了得到每个对象的访存序列，我们通过修改 Linux 内核中的页表更新操作，将所有对进程页表的更新操作信息都导出来。在解析访存序列的时候，动态重构页表，通过查询这个重构页表可以得到该访存是否为页表访存、对应的进程标识（pid）和虚拟地址等信息。同时我们还收集进程运行过程中的对象内存区域信息，从而能够识别出每个访存请求来自于哪个对象。

实验结果表明，本文提出的软硬件结合的访存行为分析方法能够准确地获取真实系统上的函数级和对象级访存行为，而引入的开销很低：对于函数级划分，平均开销为 62%（与之相比，纯软件使用 Pin 方法需要 10.4 倍的开销）；对于对象级划分，平均开销仅为 1.6%。

本文的组织如下：第二节概述访存踪迹收集工具 HMTT3 的设计和实现；第 3 节介绍函数级内存行为分析的实现；第 4 节介绍对象级内存行为分析的实现；第 5 节介绍相关工作；第六节为总结和下一步工作。

2 HMTT3 的设计和实现

HMTT 系统[4,6]是我们自主研发的一个独立于平台的软硬件结合的访存行为监测与分析系统，通过硬件侦听内存总线上的访存信号，实时获取全系统访存物理地址序列，通过反查页表获取进程标识和虚拟地址。采用硬件侦听方案的优点是：对程序透明，不会引入额外的访存干扰，能够得到包括物理地址、读/写、时间戳、进程号、虚拟地址等丰富信息。目前 HMTT 已经实现到第 3 个版本，支持 DDR3-800（工作频率为 400MHz）, HMTT 的主页为 http://asg.ict.ac.cn/hmtt/。

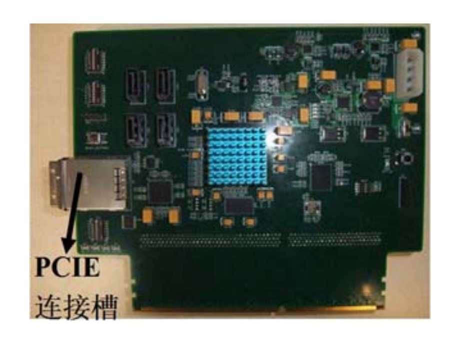


图1. HMTT3 板卡

图 1 为 HMTT3 的实物图，其上布有双列直插式存储模块（Dual-Inline-Memory-Modules, DIMM）插槽，一个负责监测内存总线信号的现场可编程阵列 Xilinx Virtex-6 LX130T FPGA，一个负责向外传输访存序列数据的 PCIE 1.0 的连接槽以及其他部件。

图 2 为 HMTT3 的内部硬件模块图。HMTT3 工作的时候，直接插在被监控机器的主板 DIMM 槽上，内存条则改插在 HMTT3 卡上安装的 DIMM 槽上，现场可编程阵列采用旁路侦听的方式监测内存命令和地址总线上收到的所有访存信号。在现场可编程阵列内部维护 DDR3 时序状态机，识别出每个读写访存请求，再通过查询预先测出的机器物理内存到 DRAM 地址的映射表，构造出此次访存的物理地址。打包单元负责将访存地址，时间戳，读写等信息组织并打成包放入 FIFO[2]队列中。PCIE 硬核负责将访存请求以 PCIE 包的格式，通过 PCIE 电缆（cable）传送到接收机上，进而保存到接收机的 Trace（踪迹）文件中。另外 HMTT3 上还包含多种配置寄存器，用户在启动 HMTT3 之前可以将配置信息写到这里以指导 HMTT3 正确工作。比如用户可以指定只监控某些特殊段的物理内存的访问（如只监控 1GB 到 2GB 之间的访存），灵活指定用于特殊标签同步访存的地址空间，指定接收机器上用于存放访存序列的物理内存起始地址和大小等。

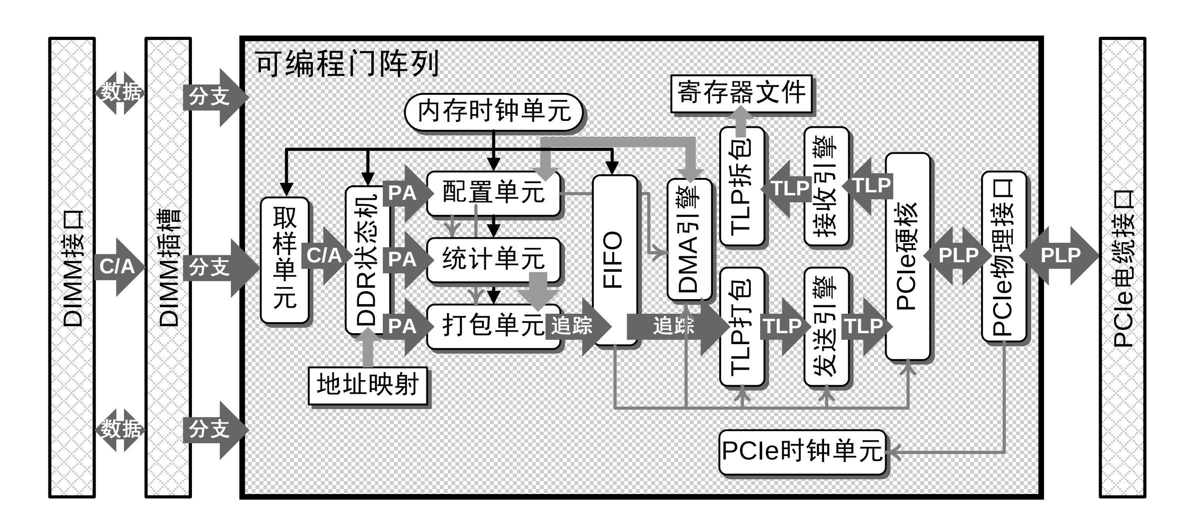


图2. HMTT3 内部硬件模块图

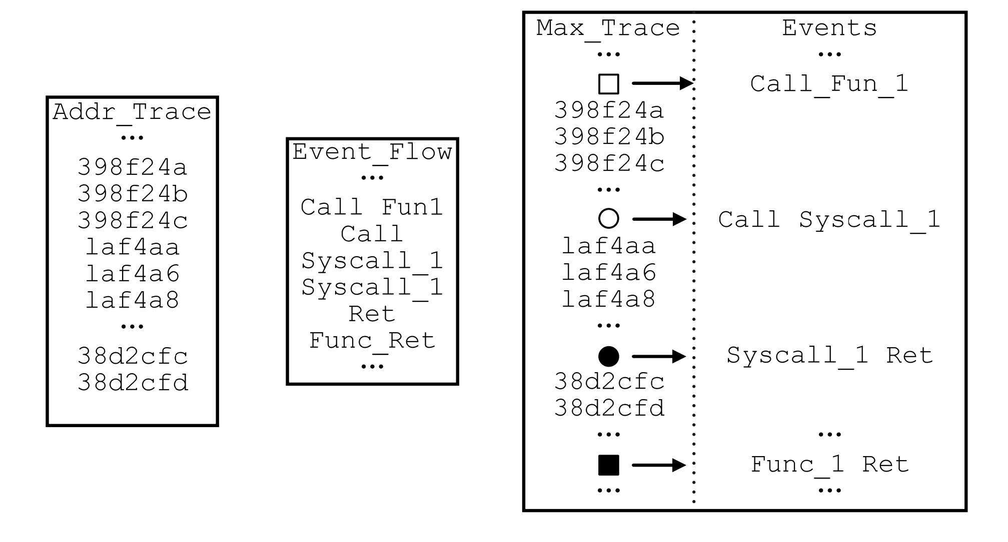
3 函数级访存行为分析

基于硬件侦听的 HMTT 卡无法直接获取程序执行过程中的高层事件流信息，这被称为硬件侦听与高层事件的语义鸿沟问题（Semantic Gap），如图 3 所示。图 3（a）为 HMTT 获取的一个访存地址序列，但不包含任何程序执行的高层语义信息。从程序员的角度来看，一个程序（或进程）的执行过程可以看作是高层事件的执行序列，比如函数调用序列、基本块执行序列，甚至精细到指令执行序列。按照高层事件流可以将一个进程的执行过程划分成不同的阶段，比如进入一个函数，从函数返回等，如图 3（b）所示。由于在不同的阶段，程序往往具有不同的访存行为特征，如果能够将底层硬件获取的访存地址流与高层事件序列进行同步并关联起来，那么我们就可以在更细粒度上分析程序在不同阶段的访存行为，比如在函数级别，如图 3（c）所示，从而对那些导致大量不规则访存的函数进行重点优化。

为了解决上述函数级别语义鸿沟问题，本文提出了一种基于二进制插桩技术的软硬件结合的方法，称之为 HMTT_FBI （Functional level Binary Instrumentation toolkit for HMTT）。通过直接修改内存中的进程映像（process image），在目标函数的入口及出口位置分别插入一条额外的特殊标签访存指令。这些标签访存与程序正常的访存都将被 HMTT 硬件侦听到。通过解析不同标签访存与函数入口、出口之间的映射关系，就能够将底层的访存踪迹与高层

[2] First In First Out，先进先出

的函数执行序列关联起来，从而实现对访存踪迹的函数级别分割。



(a)访存地址序列　(b)高层函数调用时间流　(c)包含函数级语义信息的访存地址序列

符号 □○●■ 为代表函数执行时间的特殊标签（tag）访存

图3.　访存地址序列与高层事件的语义鸿沟问题

我们通过 Linux 系统提供的 ptrace API 来获取对进程的控制权。插桩实现是在每个插桩点把原有指令替换为一条跳转指令，把对目标函数的调用指令选为插桩点，这样能够避免由于函数存在多个可能出口点带来的复杂处理。插桩代码执行完后，通过一条跳转指令将执行流跳转回到原来的调用位置。进程除了在执行到每个插桩点时额外增加的两次跳转以及执行插桩代码本身的开销外，没有引入其他的开销。而传统的动态插桩工具，如 Pin[7], DynamoRIO[8]等，则需要在程序的整个执行过程中完全控制进程，即时编译（just in time compilation）生成包含插桩代码的代码块。这种实现不仅导致比较严重的性能开销问题，还会引入比较严重的访存干扰。

3.1 HMTT_FBI 实现

3.1.1 整体框架

图 4 是 HMTT_FBI 的整体框架。这里涉及两个进程：一个运行特定的应用程序，称为被监控进程；另一个负责分析被监控进程，并向被监控进程插桩代码，称为控制进程。控制进程通过 ptrace API 获取对被监控进程的控制，支持两种模式：（1）控制进程创建被监控进程，这样允许控制进程从被监控进程执行的最开始位置就对其进行插桩，从而能够收集到被监控进程的完整信息；（2）被监控进程正在运行中，控制进程附着（attach）到被监控进程上，然后对其进行插桩，这样允许我们灵活设置只对感兴趣的运行区间进行插桩并收集信息，而不需要重新启动进程。这对于监测那些长期运行的进程是很有好处的。

控制进程主要包括 4 个模块：控制模块、反汇编分析模块、ELF[3]解析模块和 ptrace 模块，如图 4 所示。其中，控制模块向用户提供交互式操作接口，用户可以在这里指定附着（attach）到被监控进程上，获取被监控进程的状态信息（如寄存器的值、某段地址空间内的值），对被监控进程进行代码插桩，恢复被监控进程的进程映像，从被监控进程断开（detach）控制。

[3] Executable and Linking Format, 可执行与可链接格式

这些都是通过 ptrace 模块提供的接口实现的。ptrace 模块对 Linux 提供的 ptrace API 进行面向对象的封装，使用户能够更方便地对被监控进程进行操作。ELF 解析模块负责分析被监控进程的 ELF[9-12]格式的可执行文件，获取被监控进程代码段的位置和大小，通过分析 ELF 文件内的符号表（.sym）能够进一步得到每个函数的名称、入口地址及代码大小，进而获取每个函数的二进制代码。反汇编分析模块[13]则负责对得到的目标函数的二进制代码进行反汇编，主要分析函数体内每条指令的类型、位置（指令计数器值）、大小、操作数等；判断一条指令是否为函数调用指令或分支指令；如果是，进一步分析得到函数调用指令的目标函数地址或分支指令的目标地址。这些信息将被用于后面对被监控进程的代码插桩。反汇编分析模块基于一个开源的反汇编工具 udis86[14]，支持 x86 和 x86_64 指令集。

为了存放插桩代码，需要在被监控进程的地址空间内额外开辟一块空间，这通过往被监控进程的代码段内临时插桩一段 `mmap()`调用代码来实现。在插桩这段代码之前需要先将代码段内原来的内容取回保存，还需要保护进程的寄存器状态；当这段代码执行完毕，控制权将重新回到控制进程，之后恢复被监控进程的代码段内容和寄存器状态。以同样的方式，将系统的一个虚拟设备 `ioremap` 映射到被监控进程的地址空间内，这段空间将被用于作为对应高层函数事件的标签访存，其特点是对其访问不经过高速缓存，这将在第 3.1.3 节具体介绍。图 3 中被监控进程的两个阴影较深部分是我们额外再分配的两段地址空间，而阴影较浅（代码段）部分表示需要对其进行修改（通过二进制插桩实现），白色的其它部分则表示不需要做任何改动。

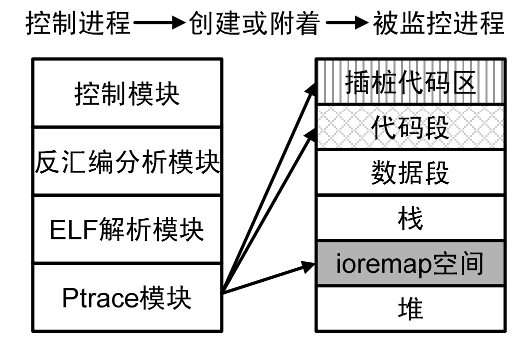


图4. HMTT_FBI 的整体框架

3.1.2 二进制插桩实现

传统的函数级别插桩都只提供对函数的入口进行插桩的接口[7-8,15]。由于函数的入口是唯一的，通过将函数体内的前几条指令替换成跳转指令就可以将控制转到插桩代码区域，从而实现收集所有与函数入口相关的信息（比如统计函数被调用次数）。而函数的出口往往不是唯一的，可能有多个。多个可能的函数出口使得直接插桩非常困难，通过复杂的指令流分析也许可以得到所有的出口位置，然后分别在这些点上进行代码插桩。这样不仅实现复杂，而且插桩的开销会变得很大。因此我们不直接在目标函数体内进行插桩。相反，我们在源头上解决问题，选择目标函数的调用指令作为插桩点，通过将其替换为无条件跳转（`jmp`）指令，把控制转移到插桩代码位置，然后将目标函数调用指令重新安排到插桩代码区域，并在该调用指令的上方和下方分别安排标签访存指令。在插桩代码区域内安排标签访存指令不会影响原程序的其他代码，可以保证程序的正确执行。

对函数的出口插桩标签访存指令是必须的。这是因为一个程序中函数之间的相互调用关系往往比较复杂，如果只标记出函数的入口，将无法判断下一个函数是在上一个函数体之内被调用，还是在上一个函数结束之后被调用。这样就无法实现对访存踪迹以函数为单位进行分割。

图 5 演示了一个函数调用的正常执行流程。当执行到函数调用（`call`）指令时，函数的返回地址（即 `call` 指令的下一条指令的 PC 值）将被压入栈顶，然后转到函数入口地址往下执行，待函数执行到返回（`ret`）指令，根据之前放入栈的返回地址，转回到函数的调用点位置之后继续执行。

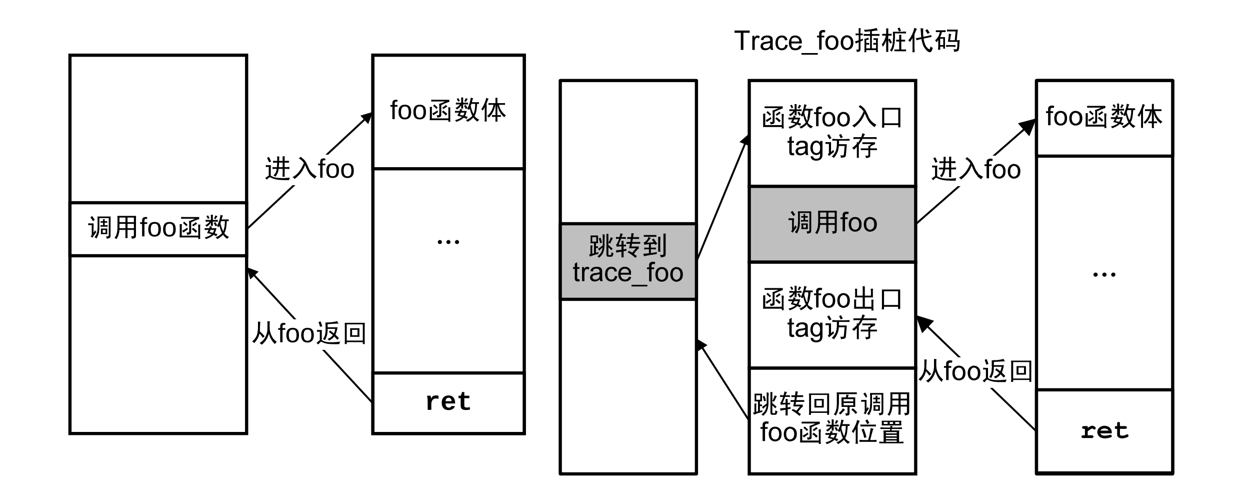


图5. 函数正常执行流程　　**图6.** 对目标函数进行插桩后的执行流程

图 6 演示了插桩后的目标函数执行流程。首先目标函数的调用指令已被替换成一条无条件跳转指令，将控制转移到对应的插桩代码区域。在这里，先执行函数的入口标签访存，然后重新调用目标函数，等到目标函数执行完毕，将返回到这个新的调用点位置。紧接着执行函数的出口标签访存，最后通过一条无条件跳转指令将控制重新转回到原来函数的调用点位置。在对函数执行额外的标签访存之前需要保护进程的状态，执行完之后恢复进程的状态，保证不影响进程的正确执行。有一点需要注意，由于此时新的函数调用指令的位置（PC 值）已经变化，而使用相对偏移的函数调用指令是根据下一条指令的 PC 值及指令内提供的相对偏移量来计算目标函数地址，所以在这里我们需要重新计算相对偏移量,以指向正确的目标函数地址。

3.1.3 标签访存

由于现代的处理器上都含有 2 级或 3 级的高速缓存，用于缓存最近访问过的数据。一旦将被访问的数据在缓存内命中，这个访存请求将不会被发送到内存系统，也就无法被 HMTT 卡采集到。由于标签访存直接对应到高层函数入口或出口事件，如果由于缓存命中导致标签访存无法被 HMTT 采集到，那么本次函数高层事件就会丢失，这样对访存踪迹的函数级别分割将不完整甚至出错。

解决这个问题的方法就是保证每次标签访存都不经过缓存，总能被 HMTT 监测到。通过修改系统的 `grub.conf` 文件配置保留一段物理内存地址空间（对操作系统不可见），使用 `ioremap` 调用将保留的这段空间映射成一个虚拟设备，并将其属性设置为不经过缓存。然后通过 `ptrace` 接口，将这个虚拟设备（命名为 `ioremap`）映射到被监控进程的地址空间内。最后建立这段 `ioremap` 空间到不同函数入口及出口标签访存的映射，就是说访问这段空间内的某个位置，将代表一个高层函数事件的发生。

使用 `ioremap` 实现的好处主要有两个：（1）对这段空间的访问不经过缓存，总能被 HMTT 采集到，不会“丢失”。（2）保留的物理内存空间对操作系统是不可见的，所以除了插桩的标签访存外，不会再受到其他的访问（如操作系统），避免受到干扰，保证采集到的标签访存的正确性和唯一性。

3.2 实验

3.2.1 实验方法

本文所用的实验平台是英特尔至强系列 E5504 服务器，包含两个处理器插槽（socket），每个插槽包含 4 个处理器核（core），工作频率为 2 GHZ。每个插槽上的高速缓存为 3 层结构，其中 L1，L2 为每个核私有，L1 的指令缓存和数据缓存都是 32KB，L2 缓存大小为 256KB，L3 缓存为同一个插槽内的 4 个核共享，大小是 8MB，每个缓存块（cache line）的大小都为 64B。物理内存使用 DD3-800，总的容量为 4GB，其中 2GB 为操作系统可用空间，另外 2GB 被保留作为 HMTT 配置空间及 `ioremap` 空间。读写（I/O）系统采用 10 个 1TB 的磁盘组成一个 `raid 0` 阵列。由于访存踪迹都是大文件，文件系统选用对大文件读写性能最优的 xfs 文件系统，使用 iozone 测试程序实测的写带宽可达 700MB/s 以上。实验平台的操作系统使用 CentOS 5.3，内核为 Linux 2.6.32。我们使用的测试程序是 SPEC CPU2006 中的访存密集型程序，编译器使用 gcc 4.1.2 版本，使用默认的-O2 优化选项。每个测试程序都使用 ref 输入集，保证实验结果的可靠性。

3.2.2 正确性验证

为了验证 HMTT_FBI 对进程注入插桩代码后，能够在每次函数调用之前及函数返回之后正确地发出标签访存，并被 HMTT 采集，进一步分析并得到包含函数执行事件的访存踪迹，我们设计了如下实验：写一些简单的测试程序，这些程序包括: （1）调用固定次数的函数；（2）多个不同函数之间相互调用；（3）递归函数。通过分析 HMTT 采集到的包含特殊标签访存的踪迹，我们能够得到不同函数之间的相互调用关系图以及每个函数被调用的次数。实验结果与直接对程序源代码的分析结果是完全一致的，这样就验证了 HMTT_FBI 工作的正确性。

进一步，我们分析更加复杂的 SPECCPU2006 内的几个程序来验证。由于这些程序包含的函数个数较多，函数调用关系复杂，我们只验证占执行时间百分比最多的前 10 个函数的被调用次数。通过 GNU[4]的 gprof 工具能够得到程序中每个函数的执行时间百分比和被调用次数。然后与通过 HMTT 访存踪迹分析得到的函数调用次数进行对比。以 `433.milc` 为例，HMTT 能够准确获取 `433.milc` 程序的函数执行事件，误差都在 2%以下，其中偏差最大的是 `mult_su3_mat_vec` 函数，HMTT 比 gprof 少了 1.90%。对其他几个程序的误差也都在 2%以下，这验证了本方法的正确性。

3.2.3 开销分析实验

图 7 说明了 HMTT_FBI 各个功能模块的运行开销，我们使用 SPECCPU2006 中的访存密集型程序，这里列出的是各模块开销占每个程序原始运行时间的百分比。可以看出，HMTT_FBI 的运行开销是很低的。其中解析 ELF 的平均开销占原程序运行时间的 0.008%；反汇编分析占 0.028%；向进程映像内插桩代码占 0.015%；整个 HMTT_FBI 的运行开销只占 0.051%，所以对原程序几乎不引入开销。其中开销最大的是 `416.gamess` 程序，达到 0.417%，这是因为 `416.gamess` 程序包含的函数多达 2840，而且每个函数所含的平均字节数达到 2621.47，这样需要解析的工作量就比较大，开销所占百分比也就比较大。

下面定量评估向进程内插入额外插桩代码导致的开销。插桩代码的运行开销包括两部分：第一个开销是插入的跳转指令，因为我们这里使用两次无条件跳转（一次从函数调用点跳转到插桩代码区，一次从插桩代码区跳转回到原执行流），每次跳转都会清空处理器流水线，导致一定的性能下降；第二个开销是插入的额外不经过缓存的 2 次标签访存（目标函数的入口和出口各 1 次），因为每次访存都需要上百个处理器周期，所以标签访存的开销是比

[4] 一个自由（免费）软件开发写作计划

较大的，如果程序本身的访存比较少，额外的标签访存将导致明显的性能下降。

为了分别评估这两个开销带来的影响，我们设计了两种实验：第一种是我们在插桩代码区内插入空指令（nop），由于空指令基本没有开销，这样我们就能评估由于跳转引入的开销；第二种就是插入完整的标签访存指令，这样得到跳转加标签访存的开销。

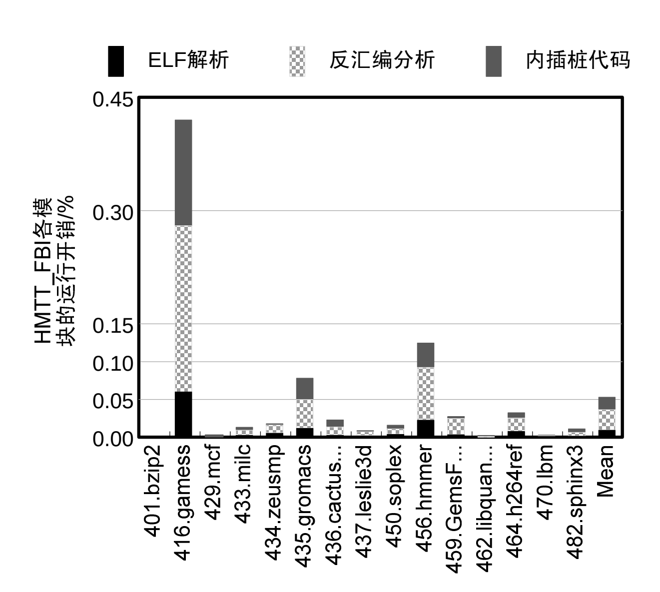


图7. HMTT_FBI 各模块运行开销

图 8 说明了这两种开销，这里将运行时间归一化到原程序的运行时间。可以看到插入跳转指令引起的开销比较小，平均为 1.042，也就是导致 4.2%的开销增加，说明使用跳转指令实现控制流转移是合理的实现。而跳转加上插入访存的平均开销为 1.62，也就是导致 62%的开销增加，其中开销最大的是 450.soplex 程序，达到 2.91。由此可见主要的开销是由不经过缓存的标签访存引起的。一个可能的改进是，通过准确记录每个函数的入口和出口的执行时间并与访存踪迹包含的时间信息进行同步，从而消除标签访存，降低开销。

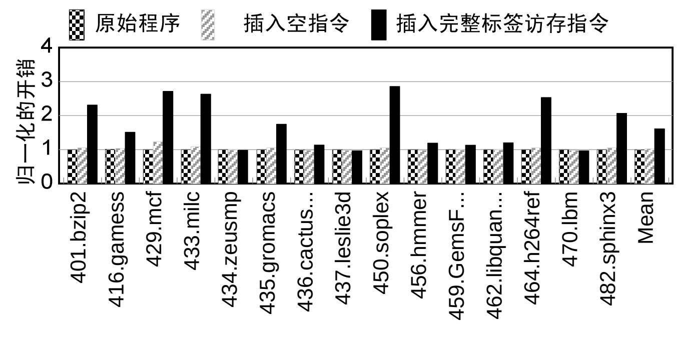


图8. HMTT_FBI 插桩的代码运行导致的开销

3.2.4 与 Pin 开销对比

使用 Pin（见注 1）纯软件方式也能获取进程执行过程中的访存虚拟地址踪迹。Pin 的实现需要在指令级进行插桩，分析每条指令是否需要访存，如果有访存，就将访存的地址记录下来，并保存到文件中。这里我们采用两种方案：第一种是先将访存地址踪迹写到内存缓冲区中，同时再开一个进程，专门负责从内存缓冲区中将访存地址踪迹写到文件中。我们采用多个缓冲区策略，保证写文件进程对 Pin 进程不会引入额外的读写开销，这样 Pin 进程只引入将踪迹写入内存的开销，称这种方法为 Pin 写到内存（Pin + Mem）；第二种也是首先将访存地址踪迹写到一个内存缓冲区内，等到缓冲区满后，再写入到文件中，这样将引入读写开销，称为 Pin 写到文件

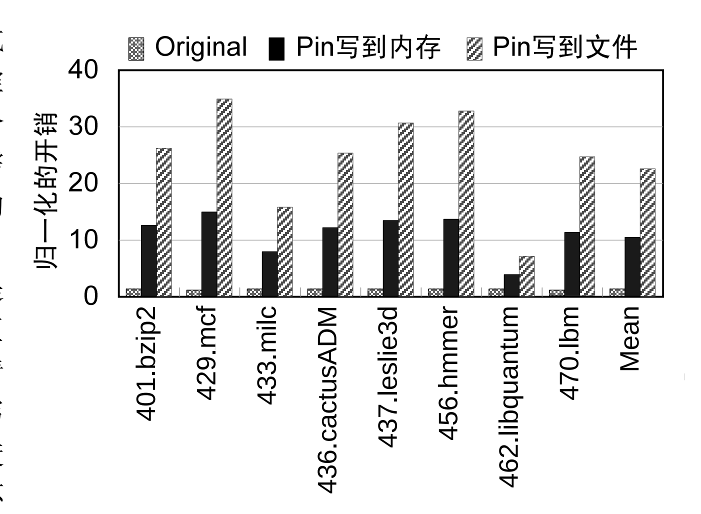


图9. 使用 Pin 获取访存踪迹的开销

（Pin + File）。注意我们这里使用的读写系统为 10 个磁盘组成的 raid 0 阵列，写带宽可达 700MB/s 以上，如果读写系统配置比较差的话，Pin 写到文件的性能将更差。

由于使用 Pin 获取访存踪迹的运行时间较长，我们只选取其中 8 个访存密集型测试程序作对比实验。图 9 列出了 Pin 获取访存踪迹相对于原程序运行时间的归一化开销。可以看到使用 Pin 的开销明显比软硬件结合实现的 HMTT_FBI 开销大，对于 Pin 写到内存，其平均开销为原程序的 10.4 倍；而 Pin 写到文件则为原程序的 22.53 倍。

4 对象级访存行为分析

一个对象代表数据存储及访问的基本单元，比如一个包含多个相同类型元素的数组，或一个包含不同类型成员的结构体。实际上对于程序开发人员而言，对象就是包含高层语义信息的数据。吴（音译，Qiang Wu）[16]等最早提出对象级访存行为分析的概念，他们将原始的访存地址序列转化为对象相关的格式，这样能够将那些具有规律访存模式的对象从整个程序的混合无规律访存序列中分离出来。图 10 演示对象级访存序列分离的过程。初始获取的为系统内不同进程的混合访存地址序列，为不规律的访存模式；下一步中我们将访存地址序列按照每个进程进行划分，可以看到这时的访存显得有些规律了，但是还是进程中多个对象的混合访存序列；最后，对每个进程的访存序列，按每个对象进行划分，这样对象级的访存模式就比较规律。获取对象级的规律访存行为，可用于多种优化，如：指导对象级的预取（Prefetch），对象级数据分配优化，对象级高速缓存划分（Cache Partition）。

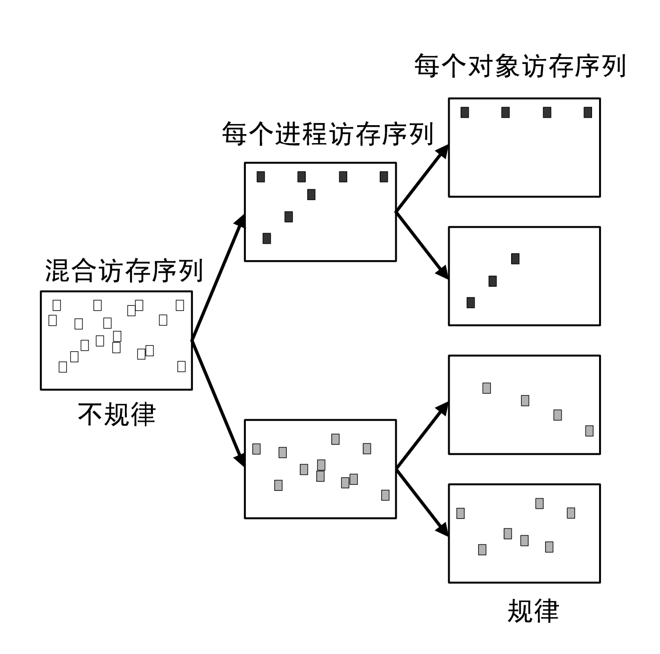


图10. 获取规律对象访存行为模式的过程

4.1 缓存实现

在本节，我们首先简要介绍软硬件混合的对象级访存行为分析的框架，然后再分别介绍其中主要模块的具体实现。

图 11 演示了软硬件混合的对象级访存行为分析的框架，主要包括 3 个模块：硬件方面，我们使用 HMTT 访存信号侦听卡，监控并收集发送到 DRAM 内存系统的所有访存请求物理地址序列；软件方面，我们修改 Linux 内核的实现，监控所有对进程页表的更新操作，并将更新操作序列导出来。这些更新序列将被用于动态重

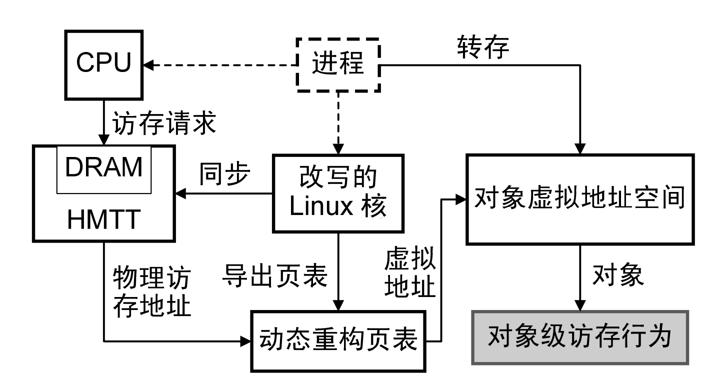


图11. 软硬件混合的对象级访存行为分析的框架

构进程页表。每个访存请求的虚拟地址通过使用 HMTT 获取的对应物理地址查询重构进程页表而得到。进一步，我们提供一个软件工具来监控并获取进程执行过程中的所有动态分配对象的虚拟地址空间信息（对象起始地址、大小）。通过比较访存虚拟地址是否处于某个对象的虚地址空间，我们能够识别出每个对象发出的访存请求序列，即将访存地址序列按对象进行细粒度划分。

4.1.1 导出页表踪迹

从 2.6.11 版本开始，Linux 内核支持 4 层基数树（Radix Tree）结构的页表[17]，它们分别是：页全局目录（Page Global Directory, PGD）、页上层目录（Page Upper Directory, PUD）、页中间目录（Page Middle Directory, PMD）和页表条目（Page Table Entry, PTE）。在 Linux 中，线性地址被分成 5 个部分，对于 x86-64 结构，常用的页面（Page）大小为 4KB，所以地址划分为（9,9,9,9,12），如图 12 所示。

Reserved (63-48)	PGD (47-39)	PUD (38-30)	PMD (29-21)	PTE (20-12)	Page Offset (11-0)

图12. X86-64 线性地址的划分

每个页目录项包含下一层页目录的物理页面号（Page Frame Number, pfn），而每个页表条目则包含存放实际数据的物理页面号以及页面的一些性质（如只读、大小等）。由于页表本身也是直接存放在内存中，实际上可以将物理内存的页面划分成两类：存放实际数据的页面（称为数据页）和存放页表本身的页面（成为页表页）。当由于执行加载（`load`）或存储（`store`）指令，而最后一级缓存发生缺失导致的访存，将被发送到数据页。而最后一级旁路转换缓冲缺失将导致至多 4 次对页表页的访存（成为 page memory walks），以取回请求虚拟地址对应的物理地址。HMTT 能够监控系统中所有的访存请求，当 HMTT 监控到一个对页表页的访存请求，就将其识别为由于旁路转换缓冲缺失导致的页表访问，而其它的访存请求则被识别为正常的数据访存。

图 13 为导出页表模块的总体结构。我们首先修改 Linux 内核中更新页表的函数，增加将更新页表操作信息导出到内核空间中的页表缓冲区中的操作。我们通过实现一个内核模块来提供对页表缓冲区的操作接口。为了实现页表踪迹和访存踪迹的同步，在监控到页表更新操作的时候，还需要向 HMTT 发送一个特殊的标签同步访存。为了灵活控制，我们还在这个内核模块提供一个控制是否启动 HMTT 监控和页表监控的接口。而用户级的控制模块进程通过向内核模块发送启动（`Start`）和停止（`Stop`）信号来实现控制，这样能够实现只对我们感兴趣的代码部分进行监控，从而降低收集的访存踪迹数据量。最后用户级的导出模块（Dump Component）则负责定期将页表缓冲区内页表更新踪迹导出到文件中保存。

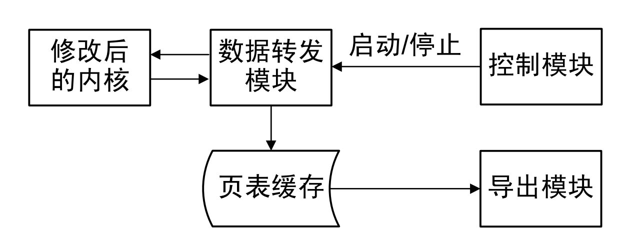


图13. 导出页表模块的总体结构

其后要解析访存踪迹，这时 HMTT 收集到的踪迹除了包含进程正常的访存物理地址踪迹外，还包含用于同步的标签访存踪迹，如图 14 所示。

图中浅色方块代表正常访存踪迹，黑色代表同步标签访存踪迹，网纹代表发送机收集到的内核（页表）更新踪迹。解析程序的工作流程如下：

(0). 从接收机得到的页表踪迹文件中，读入监控进程（指定 pid[5]）的初始页表踪迹，建立初始页表。

(1). 从踪迹文件中读出一条踪迹，解析得到物理地址、时间戳信息等。

(2). 通过查看物理地址是否落在同步标签内存区域判断得到的物理地址是否为同步标签访存；如果是，进入第（3）步，否则进入第（4）步。

(3). 解析到一条同步标签访存，从发送机收集到的页表踪迹文件中读入下一条页表更新踪迹，根据踪迹的类型和虚实页面号对页表进行相应的更新操作，进入第（5）步。

内存踪迹
分析器
取得下一条踪迹
页表踪迹
正常内存踪迹
同步标签踪迹
页表踪迹
是同步踪迹？
是
取得下一页表踪迹
否
用物理访存地址从反向页表得出虚拟访存地址
更新反向页表

图14. HMTT 解析访存踪迹和页表踪迹的流程

(4). 解析到正常访存踪迹，利用物理地址去查询当前的页表，得到对应的进程 pid 和虚拟地址信息。进入第（5）步。

(5). 如果读指针还没有达到踪迹文件末尾，向前移动 8 字节，准备读取下一条踪迹，回到第（1）；否则解析踪迹完毕，输出统计分析信息，退出解析程序。

解析程序需要维护进程的页表，这里的重构页表比较特殊的地方是：我们通过物理地址来反查得到对应的虚拟地址，等于以物理地址（页面号）为索引，所以我们将其称为反向页表（Reverse Page Table）。由于同一个物理页面可能被多个进程共享，因此我们将这些共享的进程串成一个链表，如图 15 所示。

每个反向页表项包含进程的标识符 pid 信息和对应在这个进程地址空间内的虚拟地址。解析程序得到物理地址后，计算得到物理页面号，以物理页面号索引反向页表，再依次遍历链表直到页表项内 pid 为被监听进程的 pid 为止。为了提高搜索反向页表的速度，可以将共享物理页面的多个页表项组织成 AVL 树（自平衡二叉查找树）。

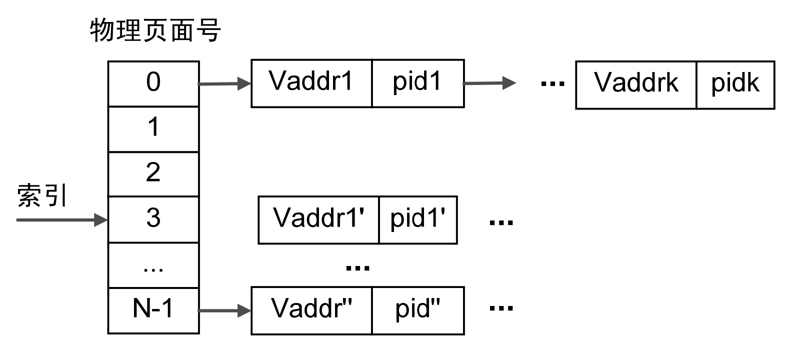


图15. 重构的进程反向页表

以物理页面号为索引，共享同一物理页面的多个进程串成链表

[5] 进程标识符

4.1.2 导出对象虚拟地址空间信息

在一个程序中，包含两种类型的对象：被分配在数据段的静态对象和程序运行过程中动态分配在堆（heap）上的动态对象。对于静态对象，可以通过分析可执行文件（在 Linux 系统中为 ELF 格式）的符号表获取它的起始虚拟地址和大小。对于动态对象，我们通过 Linux 系统提供的 `LD_PRELOAD` 环境变量来重载动态内存分配的 `malloc`(), `calloc`()等函数，以及内存释放 `free`()函数，将动态分配对象的起始虚拟地址和大小记录下来。对于提供源码的程序，我们可以进一步将对象具体对应到程序中的主要变量，从而能够识别出导致访存性能瓶颈的关键变量。对于没有源码的程序，我们可以结合程序运行过程中的函数调用栈信息来定位关键对象所处的代码段位置。

访存踪迹能够扩展成如下的格式：

<paddr, r/w, time-stamp, pid, vaddr, object, page-walk>

其中，paddr 代表访存物理地址，r/w 标识读写请求，time-stamp 为访存到达的时间戳信息，pid 代表发出本访存请求的进程标识，vaddr 为访存虚拟地址，object 为导致本次访存的对象标识，page-walk 代表本次访存是否为旁路转换缓冲缺失导致的页表访存。因此获取对象级访存序列，能用于：

(1). 将进程访存序列按对象进行划分，并分析不同对象的访存特征；

(2). 分析每个对象访存量占总的访存请求的百分比，定位关键对象，这可用于指导程序及编译器优化；

(3). 分析每个对象导致的页表访存的百分比，可用于指导旁路转换缓冲性能优化。

4.2 实验及分析

本节首先介绍我们的实验配置，然后对我们的对象级访存行为分析进行正确性验证，接着对其开销进行量化分析，最后我们通过对宽度优先搜索[6]测试程序的详细对象访存行为分析来证明本方法的有效性。

4.2.1 实验准备

本节使用与 3.2.1 节相同的实验平台，不同的是在这里我们只保留 0.25GB 的内存作为 HMTT 配置空间和页表缓冲区。因而，系统可用的实际内存空间为 3.75GB。处理器上的每个核都含有私有的两级旁路转换缓冲，L1 DTLB 含有 64 项页表项用于 4KB 页面，32 项用于 2MB 页面（称为大页面, huge-page）。第二级旁路转换缓冲含有 512 项用于 4KB 页面，没有用于 2MB 大页面的页表项。实验用的测试程序包括 Graph500 测试程序中的一个自实现的混合版 MPI/pthread 宽度优先图搜索程序[18]和 pthread 版本的 PARSEC-2.1[19]测试程序。宽度优先搜索程序默认对随机生成的 RMAT （Recursive MATrix）无尺度图（Scale Free Graph）进行 64 次宽度优先搜索，这 64 个起始顶点是随机选择的。由于本文只对单节点（node）上的宽度优先搜索进行对象级访存行为分析，即使用 1 个进程带多个线程的方式来运行宽度优先搜索，对于不同线程数目我们分别运行 3 次取平均值。

4.2.2 正确性验证

[6] Breadth-First-Search，BFS，亦有译为“广度优先搜索”

为了验证对象级访存行为分析的正确性，我们首先使用一个具有简单访存模式的微测试程序（micro-benchmark）。这个程序首先分配 5 个数组，分别命名为 a0、a1、a2、a3 和 a4，每个数组的大小为 256MB。在使用 `memset` 进行初始化后（保证系统实际分配物理页面），我们开 5 个 pthread 线程并行分别对这 5 个数组（可以看作是对象）进行不同读写比例及模式的访问，这里我们设置访问步长为 64 Byte（1 个缓存块大小）：

- a0: 全部为只读访问，正向遍历（从前往后）
- a1: 3/4 为读访问，1/4 写访问，正向
- a2: 2/4 为读访问， 2/4 为写访问，正向
- a3: 1/4 为读访问，3/4 为写访问，反向（从后往前）
- a4: 全部为写访问，反向

这里我们设置访问步长为 64B，保证每次访问都落在不同的缓存块，从而导致缓存缺失而发送访存请求。对于 256MB 大小的数组，实际的访存个数为 256MB/64B= 4M = 4,194,304。这里需要注意的是，由于最后一级缓存使用写回策略（Write-Back），所以当一个写请求发生缓存缺失的时候，最后一级缓存将首先向内存系统发送一个读请求，将要访问的数据放到对应的缓存块中，然后将新的数据写入到缓存块，等到这个缓存块被替换出去的时候（这时缓存块状态为"Dirty"），才真正发送写访存请求。所以对于每个写访问，都将导致一次读访存请求和一次写访存请求，那么对 a0~a4 的访问实际都将导致 4M 个读访存请求和不同个数的写访存请求。

表2. 微测试程序的访存分析结果

对象	读访存	写访存	比率	误差
a0	4,194,370	0	4:0	0%
a1	4,194,310	1,048,576	4:1	0%
a2	4,194,369	2,096,927	4:2	0%
a3	4,194,303	3,087,379	4:2.9	2%
a4	4,194,436	4,149,586	4:3.96	1%

表 2 为对微测试程序的对象级访存行为分析结果，其中比率指读写访存请求比例。可以看到，每个数组都大约发出 4 M 个读访存请求，这与前面的分析是一致的。对于数组 a0~a4，每个数组发送的写访存请求分别大约为 0, 1M, 2M, 3M, 4M，其中数组 a3 的误差最大为大约 2%，证明我们对象级内存行为分析的正确性。进一步图 16 演示了数组 a0 和 a4 的完整访存虚拟地址序列，可以看到数组 a0 为从前往后遍历，而数组 a4 为从后往前遍历。

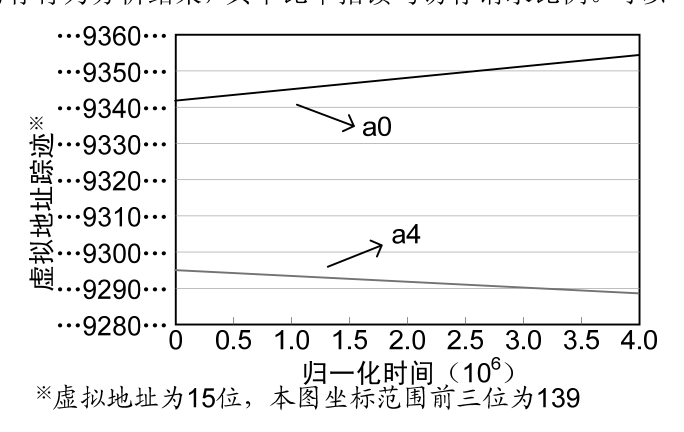


图16. 对象 a0 和 a4 的访存踪迹

4.2.3 开销分析

在本小节中，我们将评测对象级访存行为分析的开销，其中开销主要包括两部分：在内核中导出页表和导出进程的对象内存分配信息。这里我们对所有程序都运行 8 个线程来评估运行开销。

图 17 演示了对象级访存分析的开销，其中 3 个条柱 Origin, +dump_pt 和+dump_obj 分别代表程序的原来运行时间，加上在内核中导出页表信息的运行时间，以及再加上导出对象

内存分配信息的运行时间[7]。因为对大部分程序而言，主要的访存请求都来自程序中的几个主要大对象，这里我们只选择监控那些大小超过 4KB 的对象（更小的对象基本都能在缓存中命中）。可以看到，我们的对象级访存行为分析只引入很小的开销，对于内核中导出页表信息，平均只引入约 0.66%的开销；而导出主要对象的内存分配信息平均开销约为 1.60%。其中开销最大的是 dedup 程序，导出对象信息的开销达到 5.00%，这是因为对于 dedup 程序，大小超过 4KB 的对象数量达到 1,240,324 个，这么多数量的对象带来的开销自然比较大。

对于这种情况，为了进一步降低开销，我们可以选择只监控那些按大小排列居于前 10%的对象。

在程序运行过程中导出页表信息和对象内存分配信息会对缓存和内存带来一定的干扰。实验中我们发现实际上这种对内存的干扰是很小的，这是因为导出的这些信息的数据量相对于程序正常的访存数据量而言是极小的。比如对于 8 线程的 native 输入集的 canneal 程序，一共收集到 200GB 大小的访存物理地址踪迹，但是只收到 6.2MB 的页表踪迹和 26.8KB 的对象信息踪迹；同样地，对 8 线程的 streamcluster 程序，收到 104GB 的访存物理地址踪迹，而页表踪迹为 6.7MB，对象信息踪迹为 27.4KB，可见对程序正常访存引入的干扰不足 0.1%。

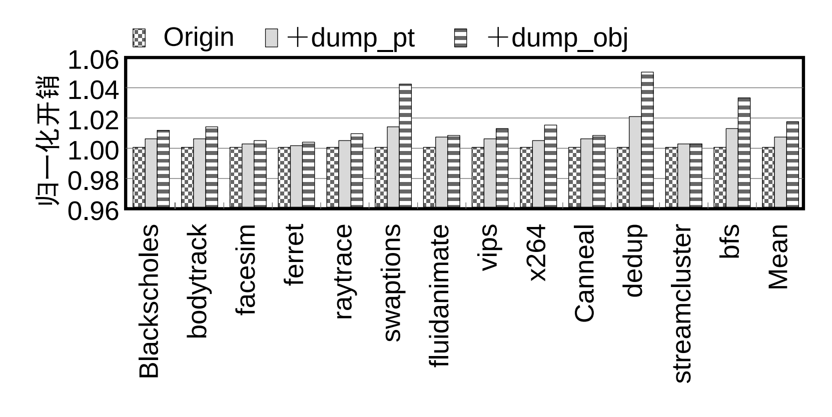


图17. 对象级访存行为分析的开销

4.2.4 Graph500 中的宽度优先搜索程序

Graph500[18]是一个比较新的针对大规模数据密集型应用，尤其是针对大规模图算法分析的测试程序集。宽度优先搜索是图算法中最重要也是最基础的操作之一，是其他多种图算法的基础。Graph 500 中的图采用克罗内科（kronecker）算法来生成，图的顶点度分布满足指数规律，即满足大部分顶点的度[8]都很小，而少量顶点具有很大的度（这也被称为小世界图）。为了节省图存储空间，图采用压缩稀疏行（Compressed Sparse Row, CSR）格式来表示。在压缩稀疏行格式中，主要的数据结构（或对象）是 `column`（列）和 `rowstarts`（行起始下标），`column` 对象连续存放每个顶点的邻接顶点阵列，`rowstarts` 对象存放每个顶点在邻接矩阵中的起始下标。在每次执行宽度优先搜索的时候，会维护一个代表顶点是否已经被访问过的位图（Bitmap）信息 `visited`（1 代表已访问过，0 代表还没被访问），一旦顶点被访问过，以后都不会再去扩展该顶点的邻接矩阵。使用 `visited` 位图能够大量减少对顶点邻接矩阵的重复访问，从而能够大大提高宽度优先搜索的性能。每次宽度优先搜索，每个顶点对应的 `visited` 对象需要被访问多次。同时在宽度优先搜索的时候还需要维护两个队列 `oldq` 和 `newq`，分别代表当前层需要扩展的顶点集合和下一层需要扩展的顶点集合，而 `pred` 对象用于存放每个顶点在宽度优先搜索树中的父顶点。在试验中，我们运行规模

[7] 因为使用 HMTT 硬件监控系统的所有访存物理地址序列几乎没有开销，这里我们没有显示

[8] 顶点的度是指与该顶点相邻的顶点个数

（scale）为 23，边因子（edgefactor）为 16（默认值）的图，即图中包含 2^{23} 顶点，每个顶点的平均度为 16。整个压缩稀疏行图需要的存储空间约为 2GB，位图 visited 的大小为 1MB，这足够放入最后一级缓存（为 4MB）。

表 3 为宽度优先搜索程序执行时随着线程个数的增加主要对象的访存个数的变化情况，表中，第一列代表线程个数，后面的几列分别代表不同对象的读写访存请求占总访存请求的比例，比如 col_r 代表 column 对象的读请求所占比例，而 col_w 代表 column 对象的写请求所占比例；total 列代表这几个主要对象的总访存占程序总访存的比例。其中 column 对象的访存占程序总访存的比例最大，在 1 个线程的时候占 65.71%，128 线程时占 54.22%，而且可以看到 column 对象为只读访问，写请求始终为 0。随着线程个数的增加，column 对象的访存所占比例在下降，而 visited 对象访存所占比例从 1 个线程的 6.08%增加到 128 个线程的 23.65%。在 total 列，可以看到对不同线程个数，超过 96%的宽度优先搜索访存都是由这些主要对象发出的，证明我们方法的正确性。

表3. 宽度优先搜索测试程序中不同对象的访存比例

线程数	col_r	col_w	visited_r	visited_w	row_r	row_w	pred_r	pred_w	total
1	65.71%	0.00%	5.21%	0.87%	7.79%	0.00%	8.43%	8.44%	96.46%
2	64.96%	0.00%	5.36%	1.32%	7.72%	0.00%	8.35%	8.35%	96.06%
4	55.19%	0.00%	16.01%	4.54%	6.57%	0.00%	7.10%	7.06%	96.46%
32	55.13%	0.00%	17.11%	5.28%	6.58%	0.00%	7.10%	7.06%	98.24%
128	54.22%	0.00%	18.38%	5.27%	6.47%	0.00%	6.98%	6.93%	98.26%

图 18 显示随着线程个数的增加，每个主要对象的访存请求（包括读和写）个数的归一化值，以 1 个线程的访存个数为基本值。可以看到除了 visited 对象外，其它对象的访存个数几乎保持不变（最大增加幅度约为 0.59%），而 visited 对象的访存请求个数在 2 线程的时候增加了 79.04%，4 线程时增加 547.81%，128 线程时增加 662.27%。这主要是因为在宽度优先搜索过程中，每扩展一个顶点都需要访问 visited 对象来确定该顶点是否已被访问过，所以对 visited 对象的访问比较频繁（实际上访问次数等于该顶点的度）。前面分析过，visited 数组的大小为 1MB，而最后一级缓存的大小为 4MB，在单线程的时候，大部分的 visited 访问都能在缓存中命中；而随着线程个数的增加，由于线程之间的相互

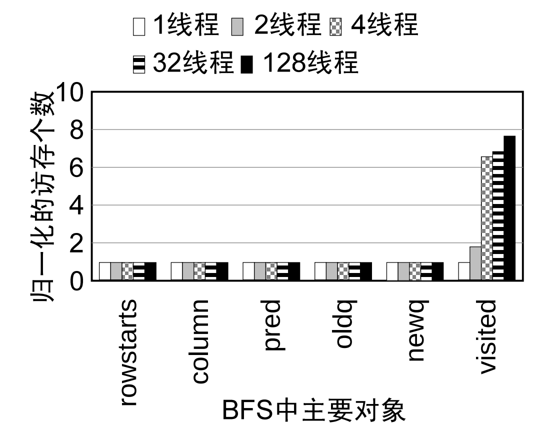


图18. 随着线程个数增加不同对象访存请求个数的变化

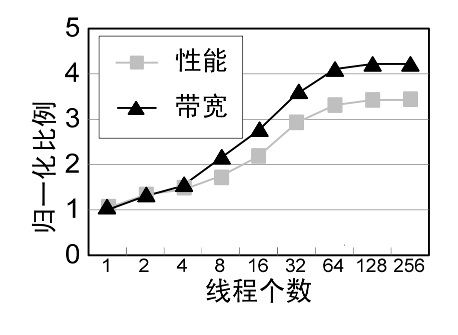


图19. 随着线程个数增加带宽和性能的变化

干扰和对共享缓存的竞争，会导致对 visited 的访问模式变得更加随机，从而产生更多的缓存缺失，导致发出更多的访存请求。而对于其他对象，由于 visited 的过滤作用，实际

上对应每个顶点的元素都只被访问一次，而且是随机访问，所以这些对象的访问模式受到最后一级缓存的相互干扰的影响很小，其访存个数几乎保持不变。

图 19 显示随着线程个数的增加，宽度优先搜索程序的性能和访存带宽的变化情况，这里的性能采用 Graph 500 内的性能指标—每秒钟访问的边个数（Traversed Edges Per Second, TEPS[18]）。可以看到随着线程个数从 1 增大到 256，性能和访存带宽都随着提高，而且存在 3 个不同阶段：第 1 个阶段从单线程到 4 线程，性能的提高和访存带宽的提高几乎一样，说明性能的提高完全得益于访存带宽。在这个阶段，线程个数翻倍能够带来约 1.2 倍的提升。第 2 个阶段从 4 线程到 64 线程，随着线程个数翻倍，访存带宽提高约 1.28，而性能提高稍低，为 1.23。这是因为在这个阶段，线程个数已经较多，导致对最后一级缓存的相互干扰加重，从而导致更多的访存。从前面的分析可知，主要是 visited 对象的访存在 4 线程之后大量增加，从而影响性能；而第 3 个阶段从 64 线程到 256 线程，这时访存带宽和性能都趋于平稳，只有约 0.018 的提高，这时因为宽度优先搜索的访存比较随机，而此时已经达到了访存带宽的瓶颈，增加线程个数不会再提高访存带宽。试验中，我们发现在 256 线程的时候，宽度优先搜索的访存带宽达到 4.66GB/s（通过分析 HMTT 访存踪迹得到），而系统的理论峰值内存带宽为 6.4GB/s，已经达到 72.81%的内存带宽使用率。由于宽度优先搜索具有比较随机的访存模式，还有线程间的锁开销，以及每次宽度优先搜索的同步开销等，这些都导致宽度优先搜索的访存带宽扩展遇到瓶颈，在这个阶段，性能同样也几乎没怎么提高（约为 0.019）。

实验中我们还发现，线程数目超过 1024，宽度优先搜索的性能会由于更加严重的最后一级缓存干扰而出现下降。所以增加线程个数，一方面能够提高访存带宽，另一方面又会导致缓存干扰加剧，尤其是对 visited 对象。所以选择合适线程个数是一个折衷。为了减低 visited 对象在缓存中被干扰，我们可以进一步使用页着色（Page Coloring）技术来为 visited 对象提供足够多的颜色，从而使其尽量在缓存中命中。

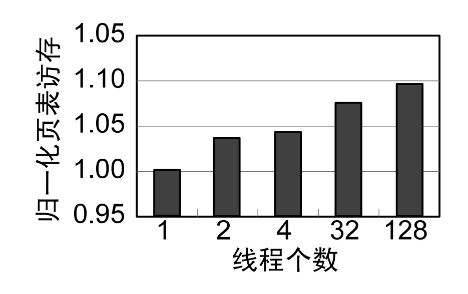


图20. 页表访存随着线程个数增加的变化

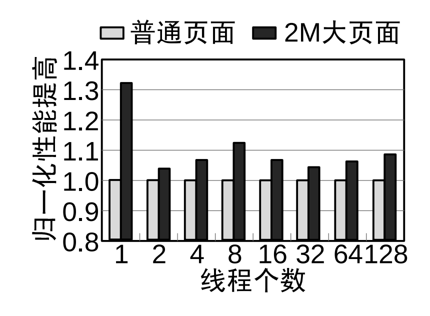


图21. 对已访问对象使用大页面优化的性能提高

图 20 显示随着线程的增加，由于旁路转换缓冲缺失导致的页表访存的归一化增长比例，相对于单线程，2 线程增加 3.5%，而 128 线程增加 9.6%。进一步分析，我们发现对于宽度优先搜索中的 visited, newq 和 oldq 对象几乎没有产生页表访存，这是因为 visited 的大小只有 1MB 左右，基本都能放在旁路转换缓冲中；而 newq 和 oldq 的访问都比较连续，几乎不会导致旁路转换缓冲缺失。其它的 3 个对象产生绝大部分页表访存, 其中 rowstarts 约占 11.92%，column 约占 72.85%，pred 约占 14.02%。column 由于本身的大小接近 2GB,而且它的访存模式为顶点内部连续而顶点间随机，所导致的页表访存量最大。

我们可以使用 2MB 的大页面来存放 column 对象，以降低其页表访存，从而降低旁路转换缓冲缺失。图 21 演示使用大页面优化相对于未使用大页面的性能提高，可以看到对于

单线程，性能可以提高 31.59%，而对多线程，性能提高没这么明显，比如对 128 线程，提高 8.18%。这是因为随着线程个数的增加，对 `column` 对象的访问变得更加随机，而用于大页面的旁路转换缓冲项数有限（为 32 个），导致更多的旁路转换缓冲访问干扰。但是另一方面，对于宽度优先搜索，增加线程个数能够更均匀地划分工作量，从而达到更好的负载平衡，降低每层宽度优先搜索的同步开销。所以可以看到使用大页面的性能提高先增大，而后又降低，在 8 线程时最好，达到 11.92%。

5 相关工作

目前国内外获取或分析访存踪迹的方式主要有以下几种：模拟器、代码插桩（instrumentation）、性能计数器、硬件仿真等。

模拟器：模拟器是进行访存行为研究的重要工具，常用的模拟器有 DramSim2[20]、Simics[21]、 Gem5[22]等，但是模拟器的速度很慢，只能用于分析很小时间段内的访存序列，无法运行完整的应用程序。一些有代表性的工作包括：MemSpy[22]，一个通过软件内存系统模拟并提供细致底层内存系统事件统计的工具，可用于定位程序的访存瓶颈；威登多佛尔（Weidendorfer）等[24]采用运行时插桩外带缓存模拟的方法来分析访存行为；SIMT[25]，一个执行驱动的主要用于内存性能研究的模拟器，能够提供多种不同的内存配置。

插桩：代码插桩通过解析可执行程序的指令流，能够得到完整的访存地址序列，但是代码插桩的开销比较大，而且引入大量的干扰访存。常用的工具有 Pin[7]、DynamoRIO[8]、Valgrind[26]、ATOM[27]、Dyninst[15]和 PEBIL[28]等。

Pin[7]是一个应用很广的动态插桩工具，工作方式是利用即时编译器动态产生最终被运行的代码，在产生代码的同时注入插桩代码。由于 Pin 的易用性、灵活性及丰富功能，使其成为学术界影响力最大的插桩工具之一。Pin 也是基于 ptrace API 来获取对进程的控制。但是使用即时编译导致 Pin 的运行开销比较大，常常导致性能下降到原来的几十分之一，而且 Pin 的内部实现是不开源的，用户很难对插桩进行优化。

DynamoRIO[8]也是一个基于即时编译器的功能丰富的动态插桩工具。由于在产生代码时加入了很多优化，其开销甚至比 Pin 要低一些。但是 DynamoRIO 是基于 Linux 系统下的 `LD_PRELOAD` 环境变量将动态共享库加载到进程空间，因此无法对正在运行中的进程进行插桩。同样使用即时编译尽管能够提供强大的插桩功能，其开销还是很大。

Dyninst[15]和 PEBIL[28]都是通过将插桩点的指令替换成跳转指令来实现将控制转移到插桩代码位置。其中 Dyninst 既支持在进程运行时对内存中的进程映象动态插桩，也支持向固态的可执行文件内静态插桩，生成一个新的可执行文件。但是 Dyninst 的插桩代码功能都是通过函数调用实现，对于那些实现简单功能的插桩，开销比较大。而 PEBIL 则只能对可执行文件（为 ELF 格式）进行处理，生成一个新的包含插桩代码的可执行文件，而无法对正在运行中的进程进行插桩，缺乏灵活性。

上面介绍的这几种插桩工具都有一个缺点，就是无法对函数的出口进行有效的插桩。使用 Pin 和 DynamoRIO 可以在函数级别进行插桩，但是它们提供的 API[9]只支持对函数入口的插桩。尽管这些工具也能够以指令级别进行插桩，找到目标函数的调用指令，将插桩代码安排在调用指令之后，从而实现对函数出口的插桩，然而以指令级别进行插桩意味着即时编译

[9] Application Programming Interfaces，应用程序接口

需要对每条指令都进行解析判断，重新生成代码，开销将非常严重，由此引入的干扰访存也将大大增加。

还有其它一些使用插桩进行访存行为分析的工作，如：SIGMA[29]是一个数据收据框架，采用软件插桩的方式来收集访存地址序列，它还提供许多包含函数和变量信息的模拟和分析工具；Ephemeral[30]使用统计采样以降低插桩开销；shadow profiling[31]和 SuperPin[32]则对插桩代码进行并行化；张翔宇[33]等使用静态插桩方式收集包含访存地址序列的完整程序踪迹，并对踪迹进行高效压缩；韦恩伯格（Weinberg）等[34]采用基本块采样的方法来降低访存分析的开销，并且避免了引入明显的偏差；罗伊（Roy）等[35]提出一种混合静态和动态二进制代码插桩的方式来降低开销；马拉塔（Marathe）等[36]提出两种混合使用硬件性能计数器和软件插桩的方法来定位共享内存应用中的缓存一致性瓶颈。

硬件性能计数器：硬件性能计数器是一种很常用的分析访存行为特征的低开销工具。但是它只能得到关于访存行为的整体统计信息，比如缓冲缺失, 旁路转换缓冲缺失等，而不能获取应用的完整访存踪迹，无法对应用访存行为做更深入的分析。艾拉尼安（Eranian）[37]指出硬件性能计数器是定位和理解内存系统性能的重要的低开销手段。一些典型的工作如：伊茨科维茨（Itzkowitz）等[38]在 UltraSPARC-III 机器上使用硬件计数器进行访存行为分析，但是只能得到事件次数，为此他们进一步使用 Sun ONE Studio 编译器来支持为每条访存提供指令细节信息；巴克（Buck）等[39]在 Itanium 2 平台上开发出 Cache Scope 工具，基于硬件事件计数器采样获取缓存缺失的地址，并根据编译调试信息将访存地址关联到程序中的变量；DProf[40]是一个面向数据的分析工具，能够将缓存缺失关联到数据（即变量），而不是传统的指令，DProf 使用硬件性能计数器来采样收集访存地址序列，然后将数据划分成不同的缓存缺失类型；Oprofile[41]是 Linux 系统内集成的性能分析工具，它基于对硬件计数器的采样来收集事件信息，并能够将这些事件关联到每个函数。

硬件仿真：这种方法只能获取简化系统的完整访存踪迹，与真实系统有一定的偏差，而且硬件仿真实现成本高，不易推广。IBM、英特尔等企业都研制出硬件侦听工具用于分析访存行为，如 PHA$E[42] 、MemorIES[43]等。这些工具一般都是插在特定总线上侦听访存命令，所以移植性较差，而且无法有效输出高达几十 GB 的访存踪迹。

托雷利亚斯（Torrellas）等[44]提出了一种与我们类似的软硬件混合的方法来分析多核操作系统的缓存性能特征。但是他们使用的硬件监控卡只能针对 MIPS 总线，而不是通用的 DDR 总线。而且他们的软件监控页表部分依赖于 MIPS 结构内的软件管理旁路转换缓冲来获取物理地址到虚拟地址的映射。这种方法不适用于当前最普遍的使用硬件自动管理的页表访存的方式来填入旁路转换缓冲的 x86 架构机器。而通过监控页表更新操作的方法，不依赖于任何硬件。

第 4 节提到的吴等人提出的一种对象相关的访存行为分析，将访存地址序列按对象进行划分，能够将具有规律访存模式的对象分离出来，并用于访存序列压缩。但是他们的方法是基于插桩来获取访存序列，开销比较大。本文提出的软硬件混合方式能够准确获取对象级访存行为特征，且开销很小。

详细的对象级访存分析对性能优化很有价值，如在Soft-OLP[2]中，用缓存划分指导对象级的基于页面着色技术，来减少对象之间的缓存相互干扰。但是他们使用动态插桩方式来进行访存分析，对于10%采样引入的开销高达30~80倍。本文提出的方法能够得到完整的对象级访存序列，而引入总的开销不超过6%。

6 总结及下一步工作

准确地获取程序在真实系统上运行的访存序列，并进行更细粒度的访存行为分析是进行内存系统研究和程序优化的基础。本文提出了一种低开销的软硬件混合的函数级和对象级内存行为分析方法：硬件方面使用 HMTT 卡监控全系统访存信号；软件方面使用二进制插桩的方式，通过直接修改内存中的进程映像，在函数的入口及出口插入标签访存指令，建立标签访存与上层函数执行流的对应关系，实现对访存踪迹的函数级分割；通过监控内核中的页表更新操作，导出页表信息，同时导出进程运行过程中对象的动态内存分配信息，综合分析得到对象级的访存序列。实验结果显示了该方法的准确性，而引入的开销很低。

下一步工作包括：（1）实现只对那些访存密集型的函数进行选择性的插桩；（2）对得到的访存踪迹进行更细粒度的以基本块为单位的划分；（3）对虚拟机（如 Xen）的访存行为分析，监控虚拟机内的机器内存到虚拟物理内存（P2M）的映射，实现将虚拟机访存序列按每个客户机进行划分。

参考文献：

[1] Onur M, Thomas M. 2008. Parallelism-aware batch scheduling: enhancing both performance and fairness of shared DRAM systems. Proc of the 35th Annual Int Symp on Computer Architecture. Washington : IEEE Computer Society: 63-74

[2] Qingda Lu, Jiang Lin, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. 2009. Soft-OLP: Improving Hardware Cache Performance through Software-Controlled Object-Level Partitioning. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques* (PACT '09). IEEE Computer Society, Washington, DC, USA: 246-257

[3] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. 2008. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems* (ASPLOS XIII). ACM, New York, NY, USA: 26-35

[4] Yungang Bao, Mingyu Chen, Yuan Ruan, et al. 2008. HMTT: a platform independent full-system memory trace monitoring system. In Proceedings of the 2008 ACM SIGMETRICS international conference on Measurement and modeling of computer systems (SIGMETRICS '08). ACM, New York, NY, USA: 229-240

[5] Dan Tang, Yungang Bao, Weiwu Hu, Mingyu Chen. 2010. DMA Cache: Using On-Chip Storage to Architecturally Separate I/O Data from CPU Data for Improving I/O Performance, in *the 16th IEEE International Symposium on High-Performance Computer Architecture* (HPCA-16). Bangalore, India: 1-12

[6] 阮元, 包云岗, 陈明宇, 等. 2008. 基于硬件的内存 trace 工具——MTT 的设计与实现. 电子学报, 2008, 36(8): 1519– 1525

[7] Chi-K L, Robert C, Robert M, et al. 2005. Pin: building customized program analysis tools with dynamic instrumentation. Proc of the 2005 ACM SIGPLAN Conf on Programming Language Design and Implementation. New York: ACM: 190-200

[8] Derek L B. 2004. Efficient, transparent, and comprehensive runtime code manipulation. Cambridge, MA, USA: Massachusetts Institute of Technology

[9] TIS Committee. Tool interface standard (TIS) executable and linking format (ELF) specification version 1.2. [2012-02-07]. http://refspecs.linuxbase.org/elf/elf.pdf

[10] Asim S. A system for process checkpointing and restarting (using a core dump). [2012-02-07]. http://geocitiessites.com/asimshankar/checkpointing/report.pdf

[11] Silvio C, Matrix Z. Shared library call redirection using ELF PLT infection. [2012-02-07]. http://vxheavens.com/lib/vsc06.html

[12] 杨浩, 唐锋, 谢海斌等. 2006. 二进制翻译中的库函数处理[J]. 计算机研究与发展, 2006, 43(12):

2174-2179

[13] Susanta N, Wei L, Lap-C L, et al. 2006. BIRD: binary interpretation using runtime disassembly. Proc of the Int Symp on Code Generation and Optimization. Washington : IEEE Computer Society: 358–370

[14] Vivek T. Udis86 disassembler library for x86 and x86-64. [2012-02-07] http://udis86.sourceforge.net/

[15] Bryan B, Jeffrey K H. 2000. An API for runtime code patching. Int Journal of High Performance Computing Applications, 14(4): 317-329

[16] Qiang Wu, Artem Pyatakov, Alexey Spiridonov, et al. 2004. Exposing Memory Access Regularities Using Object-Relative Memory Profiling. P*roceedings of the international symposium on Code generation and optimization*: feedback-directed and runtime optimization, March 20-24, Palo Alto, California: 315-

[17] Daniel Bovet, Marco Cesati. 2005. Understanding The Linux Kernel, Oreilly & Associates Inc

[18] The Graph 500 List, 2011. URL http://www.graph500.org/.

[19] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh et al. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October, ACM, New York, NY, USA: 72-81

[20] Paul R, Elliott C-B, and Bruce J. DRAMSim2: a cycle accurate memory system simulator [J]. IEEE Computer Architecture Letters, 2011, 10(1): 16-19

[21] Peter S M, Magnus C, Jesper E, et al. Simics: A full system simulation platform [J]. Computer, 2002, 35(2): 50-58

[22] Nathan B, Bradford B, Gabriel B, et al. The gem5 simulator [J]. ACM SIGARCH Computer Architecture News , 2011, 39(2): 1-7

[23] Margaret Martonosi, Anoop Gupta, and Thomas Anderson. 1992. MemSpy: analyzing memory system bottlenecks in programs. In *Proceedings of the 1992 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems* (SIGMETRICS '92/PERFORMANCE '92), Blaine D. Gaither (Ed.). ACM, New York, NY, USA: 1-12

[24] Weidendorfer, J., Kowarschik, M., and Trinitis, C. 2004. A Tool Suite for Simulation Based Analysis of Memory Access Behavior. In Int'l Conf. on Computational Science ICCS: 661-668

[25] Jie Tao, Martin Schulz, and Wolfgang Karl. 2003. A Simulation Tool for Evaluating Shared Memory Systems. In *Proceedings of the 36th annual symposium on Simulation* (ANSS '03). IEEE Computer Society, Washington, DC, USA, 335-342

[26] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation* (PLDI '07). ACM, New York, NY, USA: 89-100

[27] A. Srivastava and A. Eustace. 1994. ATOM: A system for building customized program analysis tools. *In Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*. ACM New York, NY, USA: 196–205

[28] Michael L, Mustafa M T, Laura C, et al. 2010. PEBIL: efficient static binary instrumentation for Linux. Int Symp for Performance Analysis of Systems and Software. Washington, IEEE Computer Society: 175-183

[29] Luiz DeRose, K. Ekanadham, Jeffrey K. Hollingsworth, et al. 2002. SIGMA: a simulator infrastructure to guide memory analysis. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing* (Supercomputing '02). IEEE Computer Society Press, Los Alamitos, CA, USA: 1-13

[30] O. Traub, S. Schechter, and M. Smith. 2000. Ephemeral instrumentation for lightweight program profiling

[31] Tipp Moseley , Alex Shye , Vijay Janapa Reddi, et al. 2007. Shadow Profiling: Hiding Instrumentation Costs with Parallelism, *Proceedings of the International Symposium on Code Generation and Optimization*, March 11-14, 2007: 198-208

[32] Steven Wallace and Kim Hazelwood. 2007. SuperPin: Parallelizing Dynamic Instrumentation for Real-Time Performance. In *Proceedings of the International Symposium on Code Generation and Optimization* (CGO '07). IEEE Computer Society, Washington, DC, USA: 209-220

[33] Xiangyu Zhang, and Rajiv Gupta. Whole Execution Traces. 2004. Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture. December 04-08, 2004, Portland, Oregon: 105-116

[34] Jonathan Weinberg and Allan Edward Snavely. 2008. Accurate memory signatures and synthetic address traces for HPC applications. In *Proceedings of the 22nd annual international conference on Supercomputing* (ICS '08). ACM, New York, NY, USA: 36-45

[35] Amitabha Roy, Steven Hand, and Tim Harris. 2011. Hybrid binary rewriting for memory access instrumentation. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments* (VEE '11). ACM, New York, NY, USA: 227-238

[36] Jaydeep Marathe, Frank Mueller, and Bronis R. de Supinski. 2006. Analysis of cache-coherence bottlenecks with hybrid hardware/software techniques. *ACM Trans. Archit. Code Optim.* 3, 4 (December 2006): 390-423

[37] Stéphane Eranian. 2008. What can performance counters do for memory subsystem analysis?. In Proceedings of the 2008 ACM SIGPLAN workshop on Memory systems performance and correctness: held in conjunction with the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08) (MSPC '08). ACM, New York, NY, USA: 26-30

[38] Marty Itzkowitz, Brian J. N. Wylie, Christopher Aoki, et al. 2003. Memory Profiling using Hardware Counters. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing* (SC '03). ACM, New York, NY, USA: 17-30

[39] Bryan R. Buck and Jeffrey K. Hollingsworth. 2004. Data Centric Cache Measurement on the Intel ltanium 2 Processor. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing* (SC '04). IEEE Computer Society, Washington, DC, USA: 58-70

[40] Aleksey Pesterev, Nickolai Zeldovich, and Robert T. Morris. 2010. Locating cache performance bottlenecks using data profiling. In *Proceedings of the 5th European conference on Computer systems* (EuroSys '10). ACM, New York, NY, USA: 335-348

[41] J. Levon et al. Oprofile, August 2011. URL: http://oprofile.sourceforge.net/

[42] Chalainanont N, Nurvitadhi E, Morrison R, et al. 2003. Real-time L3 cache simulations using the Programmable Hardware-Assisted Cache Emulator (PHA$E). Sixth Annual Workshop on Workload Characterization. Washington : IEEE Computer Society: 86-95

[43] Ashwini N, Kwok-K M, Krishnan S, et al. MemorIES3: a programmable, real-time hardware emulation tool for multiprocessor server design [C]. //Proc of the ninth Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2000: 37-48

[44] Josep Torrellas, Anoop Gupta, John Hennessy. 1992. Characterizing the caching and synchronization performance of a multiprocessor operating system. *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, October 12-15, 1992, Boston, Massachusetts, United States: 162-174

作者简介：

陈荔城： 中科院计算所先进计算机系统实验室，博士生，chenlicheng@ict.ac.cn
崔泽汉： 中科院计算所先进计算机系统实验室，博士生，cuizehan@ict.ac.cn
包云岗： 中科院计算所先进计算机系统实验室，副研究员，baoyg@ict.ac.cn
陈明宇： 中科院计算所先进计算机系统实验室，研究员，cmy@ict.ac.cn
黄永兵： 中科院计算所先进计算机系统实验室，博士生，huangyongbing@ict.ac.cn
谭光明： 中科院计算所高性能计算机研究中心，副研究员，tgm@ict.ac.cn